Optimized Simulation Framework for Spiking Neural Networks using GPU's

Radu MIRSU, Sebastian MICUT, Catalin CALEANU, Diana B. MIRSU
Politehnica University of Timisoara, 300006, Romania
Applied Electronics Department, 300223,
radu.mirsu@etc.upt.ro, sebastian.micut@etc.upt.ro, catalin.caleanu@etc.upt.ro,
betinaiovan@yahoo.com

***Abstract*—This paper presents a hardware accelerated model of a spiking neural network implemented in CUDA C. It does a short description of the mathematical model for the neural network and presents the implementation on the GPU. Additionally, it presents three methods of further accelerating the model by eliminating excess kernel launch overhead time, efficiently using shared memory and overlapping computation with data transfer. Finally, the implementation is benchmarked against an existing C++ equivalent model.**

***Index Terms*—artificial intelligence, biological neural networks, GPU computing, parallel processing, spiking neural networks.**

I. Introduction

General purpose computing on graphic processors (GPGPU) is of increasing interest [5], [7]. The functioning of a GPU is intrinsically parallel. This has influenced the evolution of GPUs over time in the sense that more chip area is allocated for parallel arithmetic processing rather than control flow and data caching. This type of architecture creates the opportunity to have an immense processing throughput of the GPU compared to the CPU given that the application is parallel. Having less data caching (or none) the GPU has to hide the memory access latency by performing sufficient arithmetic operations while the memory transactions are pending. This is possible if the application is parallel enough to allow sufficient threads to run concurrently.

II. Spiking Neural Network Model

The topology of the neural network is volumetric. Neurons are distributed along several bi-dimensional layers. Each individual neuron can be implemented using one of the following mathematical models: "integrate and fire" (I&F), "integrate and fire with burst" (I&FB), "integrate and fire with adaptation" (I&FA) and "resonate and fire" (R&F). The following paragraphs briefly describe the four models for a clearer understanding of their computational workload.

Eq. (1) describes the differential model of an I&F spiking neuron. The synaptic current I_{syn} is accumulated inside a lossy integrator (with loss factor K) as membrane potential p. When the potential exceeds threshold Th the neuron discharges and generates a spike at the output u. Further information on I&F models can be found in [9], [10], [15] and [16].

$$\begin{aligned} &\frac{\partial p}{\partial t} = I_{syn} - Kp && p \le Th \\ &p = 0 \quad u = 1 && p > Th \end{aligned} \tag{1}$$

Eq. (2) is the mathematical model of the I&FB neuron. Compared to the I&F model an additional calcium current I_{Ca} is present. The calcium current increases at low values of the membrane potential. On the other hand I_{Ca} decreases during periods when the membrane potential is high (e.g. periods with frequent spiking). It is worth noticing that the calcium current is invalidated by the Heaviside function H when the membrane potential is below p_{Ca}. This model allows the calcium current to increase during the resting period preceding a spike and then partially recover during the refractory period succeeding a spike. As a result, the neuron generates a burst of several spikes.

$$\begin{aligned} &\frac{\partial p}{\partial t} = I_{syn} - Kp + I_{Ca} H\left(p - p_{Ca}\right) && p \le Th \\ &p = U_{\_} \quad u = 1 && p > Th \\ &\frac{\partial I_{Ca}}{\partial t} = \begin{cases} -I_{Ca}^{\max} \dfrac{I_{Ca}}{\tau_h}, p > p_{Ca} \\ \dfrac{I_{Ca}^{\max} - I_{Ca}}{\tau_m}, p < p_{Ca} \end{cases} \end{aligned} \tag{2}$$

Eq. (3) is the mathematical model of the I&FA neuron. Similarly to the I&FB model, the model contains an additional current called I_K^{Ca} (calcium activated-potassium current). The difference is that I_K^{Ca} is independent of the membrane potential. The current increases immediately after each spike generated by the neuron (t_{sp} is the time stamp of the spike) and then decreases exponentially with rate τ_K^{Ca}. Threshold p_K^{Ca} is lower than most values of the membrane potential.

$$\begin{aligned} &\frac{\partial p}{\partial t} = I_{syn} - Kp + I_K^{Ca}\left(p_K^{Ca} - p\right) && p \le Th \\ &p = 0 \quad u = 1 && p > Th \\ &\frac{\partial I_K^{Ca}}{\partial t} = \frac{I_K^{\max}\delta\left(t_{spk}\right) - I_K^{Ca}}{\tau_K^{Ca}} \end{aligned} \tag{3}$$

This work was partially supported by the strategic grant POSDRU 6/1.5/S/13, Project ID6998 (2008), co-financed by the European Social Fund – Investing in People, within the Sectoral Operational Programme Human Resources Development 2007-2013.

This work was supported in part by the grant CNCSIS – UEFISCDI PNII – IDEI Grant No. 599/19.01.2009.

Digital Object Identifier 10.4316/AECE.2012.02011

As a result, the potassium current I_K^{Ca} is an inhibitory term. Whenever the neuron fires at a high rate the potassium current increases and inhibits the neuron, preventing it from over-saturation (adaptation).

Eq. (4) is the mathematical model of the R&F neuron. The membrane potential p is a complex number that oscillates with frequency ω and attenuates towards the zero (resting potential). This neuron model is sensitive to pulsed synaptic stimulus that resonates with the internal frequency of the neuron ω. If the synaptic stimulus is pulsing at a frequency that is outside the filtering range, the neuron will not fire. Further information on I&FB, I&FA and R&F models can be found in [11], [12], [13] and [17].

$$\frac{\partial p}{\partial t} = aI_{syn} + (b + j\omega)p \qquad \mathrm{Im}(p) < Th \tag{4}$$
$$p = jTh \text{ or } p = 0 \qquad \mathrm{Im}(p) \geq Th$$

Neurons are interconnected by dynamic synapses. Each synapse produces gain and delay when propagating a spike. The delay of the synapse is constant and must be set at the beginning of the simulation. The gain G^n is computed according to eq.5, 6 and 7, where $[x^{n+1}\ u_{SE}{}^{n+1}]$ is the internal state of the synapse and A is a constant gain. More information on the modeling of dynamic synapse can be found in [8], [9] and [14].

$$G^n = A * u_{SE}^n * x^n \tag{5}$$

$$x^{n+1} = 1 + \left(x^n - x^n u_{SE}^n - 1\right)\exp\frac{-\Delta t_n^{n+1}}{\tau_{rec}} \tag{6}$$

$$u_{SE}^{n+1} = U_{SE} + u_{SE}^n\left(1 - U_{SE}\right)\exp\frac{-\Delta t_n^{n+1}}{\tau_{facil}} \tag{7}$$

Some neurons are inhibitory. This means that their spikes will have a negative effect upon the membrane potential of the neuron that is receiving the spike. This is equivalent to a negative synaptic gain. Input signals can be injected in various places of the neural network. Signals act as analog sources that affect the membrane potential of neurons in the same way spikes do. The only difference is that input signals have no propagation time. The neurons where input signals are injected can be selected by the application.

Implementing the model on the GPU can produce significant speedup by simulating the model in parallel. However, in order for this to be possible the model itself needs to be parallelizable. In the case of the neural network presented above, during each step of the simulation, the membrane potential and the output of each neuron needs to be computed. A closer examination of eq. (1-4) shows that computing the new membrane potential of one neuron requires the availability of the following data: the current membrane potential of the neuron, neuron and synapse constants, recorded trace of the spike neural activity, input signal. It can be noticed that computing the membrane potential of one neuron does not need any data that is generated during the simulation of other neurons during the same time step. Therefore, neurons can be simulated in parallel.

It is worth mentioning that using CUDA/GPU is our second attempt to simulate the model in parallel. The first attempt was to distribute the model on a network of computers [18]. It succeeded to prove that speedup can be achieved by parallelizing the model. However, the speedup was small because of the high network communication times.

III. Simulating On The GPU

A. Parallel processing using CUDA [6]

Parallel processing on the GPU is done by launching kernels. Launching a kernel means initiating the execution of multiple instances of the same code. Each instance of the code runs on a different execution thread which ideally corresponds to a different execution unit. In practice, however, the number of execution units is significantly less than the maximum number of allowed threads. This means that threads wait in a queue until an execution unit is available. The identity of the thread is known inside the kernel code due to a set of variables that are generated at run time. In this way, different kernel instances can compute different things by using branches inside the code and more importantly can access different memory locations.

The GPU contains several Stream Multiprocessors (SMs). Each SM has eight Stream Processors (SPs). Threads can be grouped into blocks with up to 512 threads per block and up to 64K blocks for each kernel launch. Having two ways of organizing parallelism (blocks and threads) creates control over the way the workload is distributed inside the GPU. When a kernel is launched it is assured that all threads in the same block will be executed inside the same SM. This allows the threads in the same block to: communicate (fast), synchronize and quickly switch register context when toggling between threads.

B. The Software Model

Our proposed GPU model is object oriented and contains 6 classes as follows: **SpikingNeuralNetwork**, **Neuron**, **DelayLine**, **Synapse**, **InputSource**, and **ActivityRecorder** (classes and objects are printed in bold). Each class contains all necessary host and device (GPU) methods. Host methods serve in: create/initialize the model; send/retrieve the model to/from the device; save results. Device methods perform the simulation. Fig. 1 presents the CUDA C model. The arrows show the flow of information inside the model. The simulator sends input signals to the **InputSource** object, design parameters to the **SpikingNeuralNetwork** object and reads results from the **ActivityRecorder** object. The simulator also creates control variables and synchronizes the simulation. The dotted line marks the fact that the arrays of **Neuron** objects and **Synapse** objects are internal components of the **SpikingNeuralNetwork** object. The **DelayLine** object is responsible for keeping track of spikes while they propagate (with delay) between neurons. Consequently every **Neuron** object has a **DelayLine** pair-object.

The model is created and initialized by the host program. In order for the simulation to run on the device GPU a copy of the model needs to be transferred to the device. After the simulation is done the device model needs to be transferred back to the host in order to update the state of the host

model and to retrieve the simulation results. Transferring the model between the host and device raises a problem which is discussed and solved by the following section.

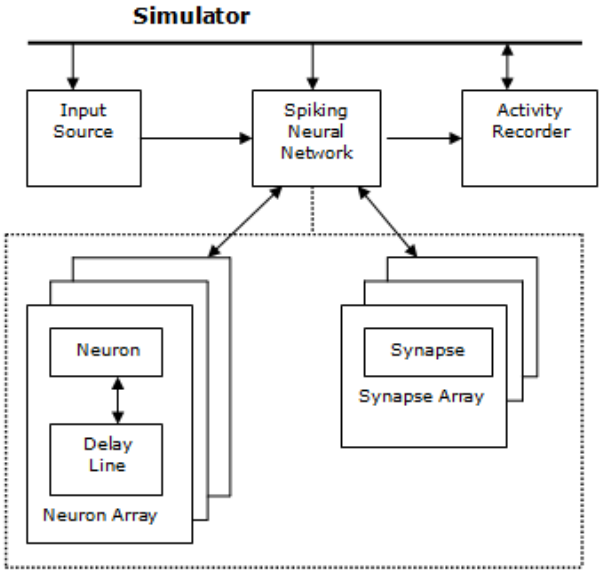

Figure 1. Model Architecture

C. *Moving Objects between the Host Computer and GPU*

CUDA C offers a single function for transferring data between the host and device (*cudaMemcpy*). The function transfers a block of data of given size between two specified memory addresses each belonging to the host and to the device respectively. In many situations, especially when memory is allocated dynamically, complex data structures (objects) are not a continuous block of data. The internal components of an object could be stored inside disjointed blocks of memory and be linked by pointers. In this case using *cudaMemcpy* directly on the object could lead to an incomplete transfer and also to false references inside the device memory. Fig. 2a presents such a situation.

One problem is that the contents of the array are not transferred. The second problem is that the pointer to the array on the device contains the memory address of the host array. If the device code tries to dereference the pointer it will cause a runtime error. Therefore, it is desired to create a standardized mechanism that allows transfer of objects from host to device regardless of the number of class layers and the arrangement of child classes within parent classes. We propose the introduction of the **BasicObject** class for this purpose. The **BasicObject** class contains all the necessary methods, memory address maps and control variables that facilitate the correct transfer of the object itself and all its internal sub-objects. All classes used in a CUDA project inherit the **BasicObject** class hence benefit from all the transfer methods. However, in order for the transfer methods to work, the new class needs a suitable constructor that correctly initializes the memory address maps and the control variables. Fig. 2b presents the concept described above. It can be seen that the **BasicObject** class does not contain the memory maps and control variables explicitly. Instead, it contains a pointer to an external structure that contains them. This is useful because it avoids transferring this information to the device (where it is not needed; both transfers host-device/device-host are performed by the host) and so it saves device memory space. The **BasicObject** class is also described in [2].

D. *Simulating the Model*

Fig. 3 illustrates how the GPU simulates the spiking neural network in parallel. Assuming that the memory has already been allocated and the model has been transferred to the GPU a set of four kernels is launched: *Propagate Synapse, Inject Input, Evaluate Neuron and Update Neuron State* (kernels are illustrated in a doubled line box). These kernels are launched sequentially (serially) by the host computer. The *Propagate Synapse* kernel reads the output of

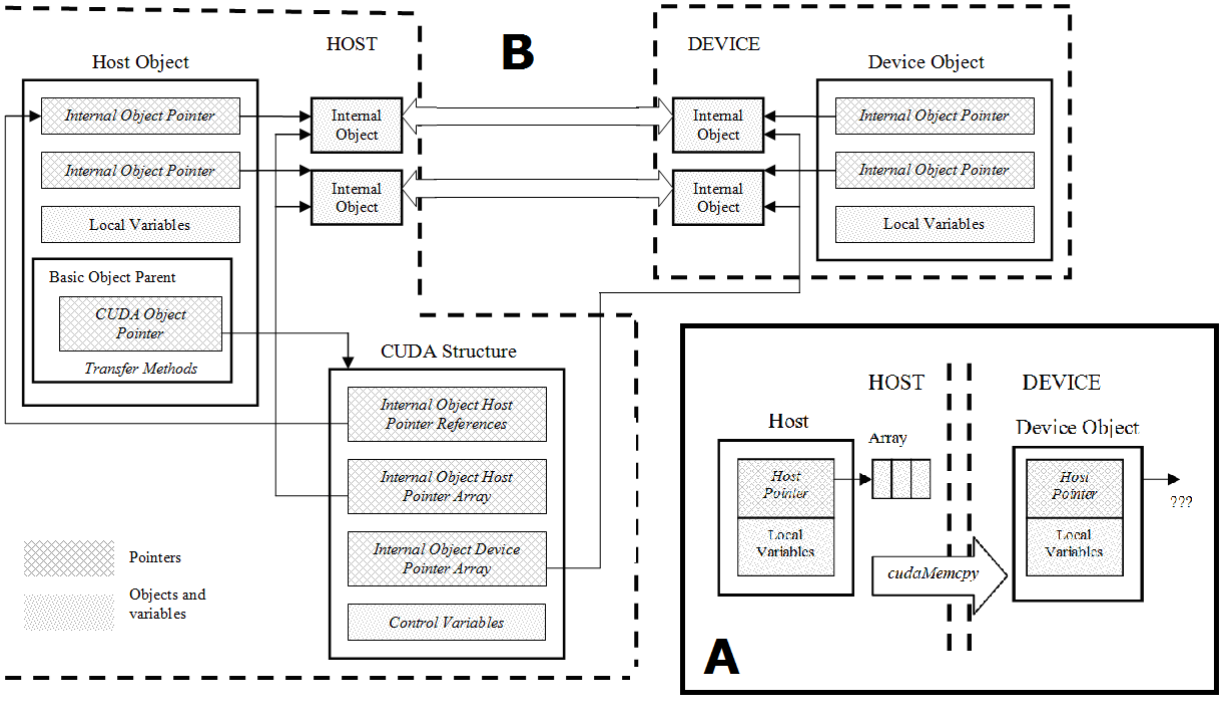

Figure 2. BasicObject Class

the source neuron. If a spike is present at the output, it amplifies the spike with the synapse gain and places the spike inside the delay line of the target neuron at a position dictated by the synapse delay. The kernel is launched on a number of threads equal with the number of synapses in the neural network, simulating in parallel as many synapses as supported by the available resources.

The *Inject Input* kernel reads the input signal and injects the signal as membrane potential inside the neurons. This kernel is launched on a number of threads equal to the number of signal injections specific to the neural network. The first two kernels are illustrated in parallel in Fig. 3. This means that the two operations do not depend on each other and their order is not important. The *Evaluate Neuron* kernel computes the next state of the neuron (membrane potential and neuron output). This kernel needs to be processed after both pervious kernels have finished execution assuring that all spikes have propagated and all input signals have been injected. This is symbolically represented by the synchronization barrier sync1 in Fig. 3. The last kernel, *Update Neuron State* updates the state of the neural network by loading the new outputs of all neurons and shifting the contents of the **DelayLine** objects. This kernel also has to be launched after all threads of the previous kernel have completed execution. The host repeats the above steps for all time steps in the simulation.

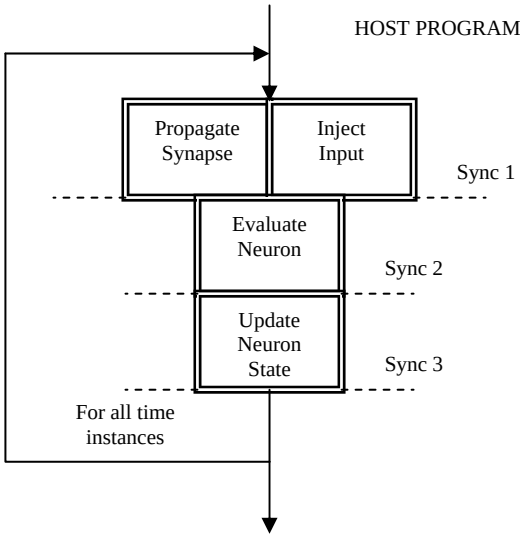

Figure 3. GPU Simulation Model

IV. Improvements To The GPU Model

A. Merging Kernels

Implementing the three synchronization barriers illustrated in Fig. 3 is done naturally using the approach presented in III.D because the host launches the kernels in a particular order and synchronizes the launch of the new kernel with the termination of the previous one (synchronization is performed by the host). However, this approach has two major disadvantages. The first one is that very many kernels are launched during one simulation. For every kernel launch an overhead time of around 3us is present. This is the time necessary for initiating the kernel execution and depends very little on the number of parameters passed to the kernel. Additionally, the overhead time does not depend on the amount of computation performed by the kernel.

If the kernel does not do a lot of computation the overhead time can represent a significant fraction out of the total simulation time. For example, if a 10s simulation is run with a 1ms time step the amount of overhead time is 10s*103(time steps/s)*4(kernels/time step)*3us (overhead time/kernel) = 0.12s, which in the case of the spiking neural model can represent up to 30% of the entire simulation time.

The second disadvantage comes from the inability to efficiently use shared memory. Because the GPU global memory is not cached, the shared memory is an alternative to accelerate data access. Each SM inside the GPU has 16KB of fast memory that is visible and can be shared by all threads of the same block. The shared memory can be as fast as a processor register as long as a bank conflict is not present during the simultaneous accesses of different threads. Anyway, the intended purpose of shared memory is that at the beginning of a kernel each thread copies the necessary data from global memory to shared memory. During the execution of the kernel, all intermediary results are stored inside the shared memory. The final results are copied to global memory only at the end of the kernel. This way, the shared memory works like a software managed cache; it is the responsibility of the programmer to assure data coherency between threads of different blocks. The lifetime of a variable declared in shared memory is only as long as the duration of the current kernel. This makes the model illustrated in Fig. 3 very inefficient because it is necessary to store intermediary results in global memory between kernels.

We propose a second approach to organizing the simulation depicted in Fig. 4. In this case all kernels, and also the main time loop are merged together into a single kernel. This way, the kernel launch overhead time is eliminated. More importantly, shared memory can be exploited very efficiently. However, because the simulation is no longer divided into kernels, the host cannot control and synchronize the simulation anymore. Once the kernel is launched, it is the job of the GPU to synchronize threads. First, the model was slightly reorganized such that it needs fewer synchronization barriers. In Fig. 3, each kernel dedicates one thread for the processing of each synapse, input injection and neuron. In Fig. 4, each thread processes everything that is related to the functioning of one neuron.

The kernel has two internal loops that process serially all synapses and input signals targeting this neuron. Therefore, once the execution reaches the “Evaluate Neuron” stage it is assured that the previous stages have completed. It is not important that the processing of other synapses or other input signals in the neural network might not completed yet, since they do not influence the functioning of this neuron during the current time step. The introduction of two loops inside the kernel does not impact the efficiency of parallel processing if the number of neurons is large enough to fully utilize the resources of the GPU.

Unfortunately, the synchronization barrier at the end of the simulation time step cannot be avoided. The hardware offers a synchronization instruction, but it is only able to synchronize threads of the same block. The only way to use

this hardware implemented synchronization is to put all threads into the same block, but that would waste GPU resources since only one SM would be used. An alternative, which allows synchronizing all threads, is to use a software implemented barrier described by the following steps:

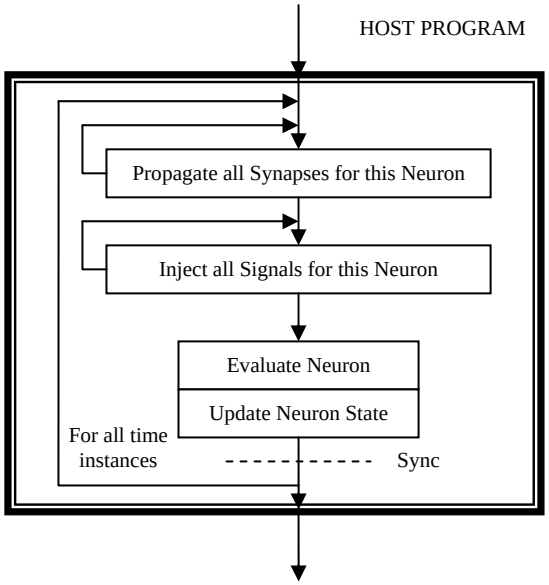

Figure 4. Unified Kernel Approach

- Synchronize all threads in each block.
- First thread of each block atomically increments counter variable that resides in global memory.
- First thread of each block waits in loop until counter reaches the total number of blocks.
- Synchronize all threads in each block.

The synchronization is done in two steps. First, all threads in each block are synchronized in hardware. Second, each block delegates a thread to communicate globally and synchronize blocks. Because the counter is a shared resource, it needs to be incremented atomically (serialized by hardware) in order to eliminate race conditions and miscounting. Also, because atomic operations are slow and because writing to global memory is slow in general, the two step approach presented above is much better than synchronizing all threads globally. However, there is a trick to this approach. When a kernel is launched the blocks are enumerated and distributed evenly to the SMs of the GPU. After counting the necessary resources for the execution of each block, it is decided how many blocks can be launched in parallel on each SM. If the SM cannot accommodate all its designated blocks at the same time, the remaining blocks wait in a queue until a running block finishes execution and resources are freed. Note that a block does not suspend execution if it is not doing anything (waiting in a loop). This can lead to a dangerous situation where the counter condition is never achieved. This is because inactive blocks wait in a queue for the active blocks to complete execution. In the meanwhile, the active blocks never complete execution because they wait for a condition that needs all blocks to be running. One way to solve the situation is to set the number of blocks equal to the number of SMs which in the case of the GT9800 GPU (employed in this paper) is 14. Afterwards, the number of threads in each block can be computed in order to have a total number of threads equal or higher to the number of neurons in the neural network.

B. Using Shared Memory

As presented in the previous paragraph, shared memory is a powerful tool to speed up memory access. Merging all kernels into a single one makes shared memory even more appealing because variables will be present in shared memory during the entire simulation without having to load/store them from/to global memory. Because the shared memory is a limited and precious resource it is important to see what to store in there and what not. A statistical analysis of how memory is accessed during a simulation reveals that almost 80% of all accesses are performed to the data **DelayLine** objects. The number of accesses to this data structure is not constant and depends on the contents of the delay lines during the simulation. However, in all situations, they dominate the overall amount of accesses, making the data **DelayLine** the best candidate to occupy shared memory. The next best candidate is the **Neuron** object which is responsible for about 14% of all accesses. This object, however, is not suitable for shared memory because the neuron outputs need to be broadcasted globally at the end of each simulation time step. This is because the model has no restrictions regarding the connectivity of the network, and so synapses can connect neurons that run on threads residing in different blocks on the GPU. Accesses to other variables are less significant in number. Therefore, in our implementation the data **DelayLine** is the only object that is stored in shared memory. In addition, the state variables I_{Ca} and I_K^{Ca} that are specific to the I&FB and I&FA neuron models are also stored in shared memory.

In order to have a higher bandwidth, shared memory is divided into 16 banks that can be accessed simultaneously as long as there is no bank conflict (two threads accessing memory locations of the same bank). The shared memory space is organized such that consecutive addresses are found in consecutive memory banks rather than the same bank. This is because in most CUDA programs successive threads access consecutive memory addresses. With this argument in mind, we propose two ways to organize the data delay line buffer.

Consider that TN is the number of threads running in the same block (and also on the same SM) and that DS is the size of each delay line buffer. Therefore, a total space of TNxDSx8 bytes needs to be allocated in shared memory for each block. 8 bytes are needed for each entry of the delay line buffer because two 32bit words need to be stored (one float word for the spike amplitude; one unsigned integer for the spike timestamp). The size of the delay line DS is constant and is set at the beginning of the application. For example, if a network of 2000 neurons must be simulated, the computing grid will have 14 blocks x 143 threads. Given the maximum of 16KB of shared memory per block the maximum delay size DS is 14 (143x14x8B = 15.64KB). The limit of the buffer size would not be present when using global memory. However, it is a compromise worth taking because the speed improvement is significant and it is rarely the situation when more than 5 entries of the buffer are needed. Anyway, if a neuron receives a spike and its delay line is already full, the spike that has the largest timestamp is eliminated.

Fig. 5 illustrates two proposed ways of organizing the delay lines. For simplicity, the figure only illustrates 4 neurons (threads) each having a delay line buffer of size 6.

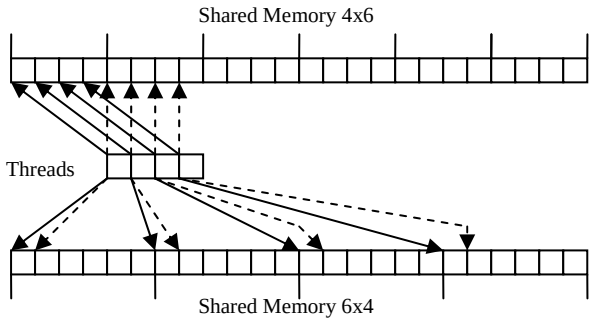

Figure 5. Delay Line Structure in Shared Memory

In the upper part of the figure the delay line entries of different neurons are interleaved. In the lower part of the figure all entries of the same neuron are grouped. In the upper part, banks conflicts are eliminated naturally because the threads access simultaneous consecutive addresses thus distinct banks. In the lower part bank conflicts are eliminated if DS%16 is odd. For this implementation the first approach was preferred because no restrictions are imposed on DS.

C. Overlapping Computations and Data Transfer

The simulator records traces of the neural activity for all neurons. At the end of the simulation two sets of traces are available: one binary set for the neuron outputs and one floating point set for the neuron membrane potential. This data needs to be copied from the device memory to the host computer memory in order for it to be available to the host main application (unless the application continues on the GPU). If the neural network is large and/or if the simulation length is large the amount of data that needs to be transferred is significant and can add up to tens or even hundreds of MB. There are situations when the host application only requires the neuron output binary traces and not the membrane potentials. If so, the amount of data that needs to be transferred is significantly reduced. Anyway, in order to reduce overall application time, NVIDIA offers hardware mechanisms that allow overlapping of computation and data transfer. This enables the user to transfer available simulation results while new results are being computed.

The NVIDIA architecture is organized as a stream processor. When the host computer expects that the device performs a certain task it places a request for this task in a queue called stream (usually a task is either a kernel launch or a memory transfer operation). The host continues executing the host program until another device specific instruction is met. At this point, the host checks if the device has completed the previous action. If true, it sends a new request, else it waits for the device to finish. This is called synchronous operation mode, in the sense that the host is always synchronized to the device. It is worth mentioning that in this type of operation mode only one stream is used and that stream will only contain one action in its queue, since no new task can be send to the device until the previous one is finished. Another type of operation is the asynchronous mode. In this mode, the host does not need to synchronize to the device until a specific synchronization instruction is reached. This way the host application can send several tasks to the device without actually waiting for them to complete. Also, the host can place the tasks on distinct streams.

Fig. 6 illustrates how this procedure allows overlapping of computation and data transfer. Consider that the simulator presented in paragraph III.D (Fig. 4) breaks the simulation into N sequential parts and launches N consecutive kernels. This is useful because as soon as the first kernel finishes execution the hardware starts to transfer the results from the first kernel while the second kernel computes.

The streams in Fig. 6 are conceptual and only exist from the programmer's point of view. In hardware, the tasks are sent to the actual engines in the same order they are inserted by the host into streams. Additionally, the hardware has an inter-engine mechanism which assures that all tasks coming from the same stream are performed in the same order as specified by the stream.

For example, task CPY0 will start after task SIM0 is completed even though they are performed by different engines. As presented in paragraph III launching a kernel produces about 3μs of overhead time regardless of the amount of computation inside the kernel. Therefore, splitting the simulation into N segments, and so launching N kernels instead of one, adds a penalty time equal to N*KLT, where KLT is the kernel launch time.

On the other side, the transfer time TRT reduces to its N^{th} fraction. Because the penalty increases linearly while the

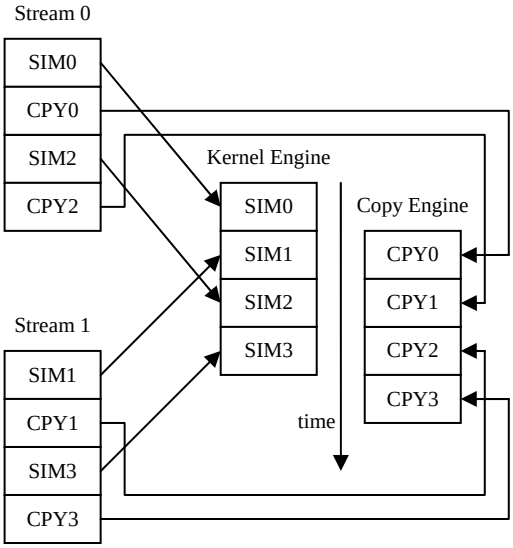

Figure 6. Overlapping computation and data transfer

gain decreases non-linearly, it is optimal to increase N as long as the rate of decrease for the gain is higher than KLT:

$$KLT \le \frac{TRT}{N} - \frac{TRT}{N+1} = \frac{TRT}{N*(N+1)} \tag{8}$$

Because N is usually a large number (over 100) eq. (9) is approximated well by the next relation.

$$N < \sqrt{\frac{TRT}{KLT}} \tag{9}$$

D. Using Constant Memory and Texture Memory

The GPU is equipped with two special types of memory spaces: the constant memory and the texture memory. Both types of memory are cached and can provide further improvement to memory access speed. As already mentioned, the GPU does not dedicate many transistors to complicated memory management and sophisticated data caching. Therefore, the constant memory and texture memory caches must be simple. The simplicity is assured by the fact that both memory spaces are restricted to being read-only. This way, the coherency of cached data is assured implicitly and significant hardware can be excluded from the cache. The only data in the spiking neural network model that is constant is the synapse information and the general neural network parameters. The neural network parameters are few and it is easier to store them in shared memory which is faster than the caches. Because of its large size the synapse information cannot be stored in shared memory and so constant memory or texture memory can represent an alternative (Synapse data size = 16B x number of synapses; up to hundreds of MB).

The efficiency of caching depends drastically on the cache hit rate. Therefore, the address access pattern that results during the execution of an application is very important and determines the efficiency of the cache. In general, if some data is read and cached, it needs to be read again at least a few more times before it is evicted from the cache. Examining the memory access pattern for this application reveals that using the cache will be inefficient. During each simulation time step the data stored by a synapse is required only once. In addition, all synapse information is required during each simulation time step. Because the cache is only 64KB and cannot store the entire synapse data, each synapse is stored and then is evicted from the cache before it is needed again. Consequently, for this application the cache cannot be exploited so the synapse information is stored in global memory.

V. Simulation Results

Before comparing simulation times of the C++ (serial) and CUDA C models it is imperative to assure that the output of the two models is the same. This comparison is done independently for each simulation time step. Two output vectors are built containing the neuron membrane potentials of the two models. The error between the outputs is considered to be the norm of the difference vector. The maximum error ever found is $3.46*10^{-6}$. This small error is present because floating point operations are not associative due to rounding of intermediate results. This means that the order in which operations are performed matters. This order cannot be guaranteed in a parallel thread-based system. Nevertheless, the output of the serial C++ model also depends on the order in which the neurons are picked for execution. Changing this order would also result in an error of the same magnitude. Anyway, in our application the system needs to be robust and insensitive to noise of significantly higher magnitude and so this error can be considered negligible.

Table I presents the achieved speedups for different neural network sizes using the I&F neuron model. The C++ model is run on a system with the following configuration: Intel Core i7 CPU at 2.67 GHz, 4GB DDR3 RAM, 64bit Windows 7. The CUDA C model is run on a NVIDIA GeForce GT9800 GPU. The speedup is calculated as the ratio of the C++ simulation time and the CUDA C simulation time. There are four versions of CUDA implementations: V1 is the CUDA model presented in Fig. 3 without any of the additional improvements presented in this paper; V2 is the CUDA model presented in Fig. 4 where the kernel overhead time eliminated; V3 is V2 with the delay lines implemented in shared memory; V4 is V3 with the overlap of computation and data transfer. The simulation times used at computing the speedups are estimated by averaging the results of 100 simulations performed on different networks of the same size and with the same number of synaptic connections.

TABLE I. SPEEDUPS FOR THE I&F NEURON MODEL

Network Size	Speedup			
	V1	V2	V3	V4
8x8x8	x1.19	x1.42	x2.41	x2.44
9x9x9	x1.43	x1.66	x2.91	x2.98
10x10x10	x1.86	x2.14	x3.83	x4.01
11x11x11	x2.45	x2.60	x4.56	x4.96
12x12x12	x2.87	x3.25	x5.49	x6.14
13x13x13	x3.38	x3.76	x6.02	x6.84
14x14x14	x3.72	x4.08	x6.14	x7.12
15x15x15	x4.05	x4.39	x6.22	x7.23

It can be noticed that in general larger networks have greater speedups. This is because for larger networks, more computational workload allows the CUDA model to better exploit parallelism and more efficiently utilize its resources. On the other hand the simulation time of the C++ model scales linearly with the amount of computations.

Table II presents a comparison between the speedups achieved by the four different neuron models: I&F, I&FB, I&FA and R&F. Reported speedups are obtained with version V4 of the CUDA model. The speedups are not the same because each model has a different flop/byte ratio (floating point operations per memory transfer operations). Because the GPU has no cache, all transfers to/from global memory are slow. However, if sufficient threads are running and if the kernel has a high number of flops compared to the number of memory accesses, the GPU can quickly switch between threads and perform flops while memory transfers are still pending. As a result the GPU hides the high latency of global memory. This means that efficiency of the GPU, hence the speedup of the model, is higher when the model has a high flop/byte ratio. Other examples of how the flop/byte ratio influences the performance of a multi-core architecture can be found in [1], [3].

The I&FB, I&FA and R&F models clearly have a higher flop/byte ratio compared to the I&F model (state variable like I_{Ca} and I_K^{Ca} are stored in fast shared memory and are not counted as transfer operations). However, a clear relationship between the flop/byte ratios and the speedups

cannot be determined. This is because in addition to the flops of the neuron model the GPU must execute flops that are related to the functionality of the **DelayLine** objects. The number of operations performed by the delay lines is not fixed and depends on the number of spikes generated by the neural network during the simulation. In turn this depends on the architecture of the network and also on the input signals. As for table I, the speedups are estimated as an average of 100 simulations for each model. Examining the results of table II reveals that statistically the flop/byte ratio increases in this order: I&F, R&F, I&FA and I&FB. This is an effect of both model complexity but also spiking activity. As expected the I&FB (which generates bursts of spikes) has the highest flop/byte ratio as an effect of very busy **DelayLine** objects.

TABLE II. SPEEDUPS FOR DIFFERENT NEURON MODELS

Network Size	Speedup			
	I&F	I&FB	I&FA	R&F
8x8x8	x2.44	x2.94	x2,89	x2.79
9x9x9	x2.98	x3.61	x3.55	x3.42
10x10x10	x4.01	x4.88	x4.79	x4.64
11x11x11	x4.96	x6.07	x5.97	x5,76
12x12x12	x6.14	x7.56	x7.39	x7,16
13x13x13	x6.84	x8.41	x8.24	x7,98
14x14x14	x7.12	x8.78	x8.64	x8,34
15x15x15	x7.23	x9.02	x8.79	x8.52

VI. Conclusions And Future Work

The contributions of this paper are: (1) Easy design of a new neural network with dynamic spiking neurons, (2) Fast simulation of large neural networks by exploiting the stunning computational power of modern GPUs, (3) Enables users to perform optimizations by reducing the duration of iterative simulation, (4) Compares the performances delivered by the GPU when simulating four different models of spiking neurons.

As future work, it is intended to use the CUDA C simulation framework at optimizing and simulating different architectures of spiking neural networks in applications of face recognition and affective computing. One possibility is to use the Liquid State Machine architecture [19] with Gabor coefficients [20], [21], [22] as input signals. On the long run we aim towards designing a unified computing architecture that is able to perform several tasks within the same neural volume. Other image processing example using GPU spiking neural networks can be found in [4].

Because this type of artificial neural network is similar to ones in the biological systems, (e.g. mammalian olfactory systems) it can be included in bio-inspired devices like E-NOSE [23] (Electronic Natural Olfactory Sensors Emulator).

Acknowledgment

This work was partially supported by the strategic grant POSDRU 6/1.5/S/13, Project ID6998 (2008), co-financed by the European Social Fund – Investing in People, within the Sectoral Operational Programme Human Resources Development 2007-2013.

This work was supported in part by the grant CNCSIS – UEFISCDI PNII – IDEI Grant No. 599/19.01.2009.

References

[1] M.A. Bhuiyan, V. K. Pallipuram, M.C. Smith, "Acceleration of spiking neural networks in emerging multi-core and GPU architectures", IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010

[2] R Mirsu, C. Caleanu, V. Tiponut, "GPU accelerated model for liquid state machine based on spiking neurons", 17th International Conference on Soft Computing (MENDEL), Brno, Czech Rep, 2011.

[3] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, A.V. Veidenbaum, "A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors," Special issue of Neural Network, Elsevier, vol.22, no. 5-6, pp. 791-800, July 2009.

[4] M. A. Bhuiyan, T. M. Taha, R. Jalasutram, "Character recognition with two spiking neural network models on multi-core architectures," in IEEE Proc. CIMSVP, TN, pp. 29 – 34, Mar. 2009.

[5] M. Herlihy, N. Shavit, *The Art of Multiprocessor Programming* Morgan Kaufmann, Burlington, MA, 2010.

[6] NVIDIA CUDATM. NVIDIA CUDA C Programming Guide 3.1.1. 2010.

[7] D. Kirk, W. Hwu, *Programming Massively Parallel Processors* Morgan Kaufmann, Burlington, MA, 2010.

[8] R. Ananthanarayanan, D. Modha, "Anatomy of a cortical simulator", *Proc. Int'l Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2007, pp. 1-12.

[9] W. Gerstner, W. Kistler, *Spiking Neuron Models*, Cambridge University Press, 2002.

[10] E.M. Izhikevich, "Simple model of spiking neurons", IEEE Transactions on Neural Networks, Vol.14, No.6, 2003, pp. 1569-1572

[11] E.M. Izhikevich, "Resonate and fire neurons", Neural Networks 14, pp. 883-894, 2001.

[12] E.M. Izhikevich, "Which model to use for cortical neurons?", IEEE Transactions on Neural Networks (Special Issue on Temporal Coding), 2004.

[13] W. Maass, "Networks of spiking neurons: The third generation of neural network models", Neural Networks, Vol. 10, No.9, pp. 1659-1671, 1997.

[14] L. F. Abbott, T. B. Kepler, "Model neurons: from Hodgkin-Huxley to Hopfield", Editor: Luis Garrido, Lecture Notes in Physics, vol. 368, p.5-18, 1990

[15] I. Bogdanov, R. Mirsu, V. Tiponut, "MATLAB model for spiking neural networks", Proc. of the 13th WSEAS International Conference on Systems, Rhodes, Greece, 23-25 July, 2009, pp. 533-537.

[16] R. Mirsu, V. Tiponut, I. Gavrilut, "Storing information with spiking neural networks", Proc. of the 13th WSEAS International Conference on Computers, Rhodes, Greece, 23-25 July 2009, pp. 318-322.

[17] S. Wills, *Computation with Spiking Neurons*, PhD Dissertation, University of Cambridge, 2004.

[18] R. Mirsu, V. Tiponut, "Parallel model for spiking neural networks using MATLAB", *9th International Symposium on Electronics and Telecommunications (ISETC)*, Timisoara, Romania, 2010, pp. 369-372.

[19] W. Maass, T. Natschlager, H. Markram, "Real-Time Computing without stable states: A new Framework for Neural Computation based on Perturbation", *Neural Computation*, Vol. 14, No. 11, Pages 2531-2560, Nov. 2002.

[20] J. Kamarainen, V. Kyrki, H. Kalviainen, M. Hamouz, J. Kittler," Invariant Gabor features for face evidence extraction", *Proceedings of the IAPR Workshop on Machine Vision Applications*, Nara, Japan, 2002, pp. 228–231.

[21] O. Ayinde, Y .Yang, "Face recognition approach based on rank correlation of Gabor-filtered images", Pattern Recognition, vol. 35(6), pp. 1275-1289, 2002.

[22] V. Kyrki, J. Kamarainen, H. Kalviainen, "Content based image matching using Gabor filtering", *Proceedings of the Int. Conf. on Advanced Concepts for Intelligent Vision Systems Theory and Applications*, Baden-Baden, Germany, 2001, pp. 45–49.

[23] D. Martinez, E. Hugues, "A spiking neural network model of the locust antennal lobe: Towards neuromorphic electronic noses inspired from insect olfaction", *Proceedings of the NATO Advance Research Workshop on Electronic Noses & Sensors for Detection of Explosives*, Warwick, Coventry, U.K., 2003, pp. 209-234.